Project Report

Program 2: Bathroom

General Description:

The code tries to simulate a unisex bathroom, where the bathroom program creates the resources that represent the 'bathroom' itself while the enter program (run multiple times) tries to mimic a person (man/woman) trying to enter the bathroom. The bathroom program also updates the user of the current usage statistics and closes and deletes the bathroom resources on termination.

Design:

Effort has been made in the project to avoid deadlocks and race conditions under all feasible conditions- this has in fact been the highest priority while designing. Also, the programs do not do a busy wait on any variable; instead, wait durations have been handled by the availability (or non-availability) of semaphores.

There are two semaphore variables in the code:
bathroom- a counting semaphore that has a value equivalent to the size of the bathroom (received as a command line argument).
gender- a binary semaphore that allows access to a variable that decides the gender of occupants that will be allowed into the bathroom.

These semaphores and some other globally available variables are stored in the following structure:

```
typedef struct semaphore_str
{
sem_t bathroom;  //bathroom semaphore- counting semaphore- upto size of bathroom
sem_t gender;   //current bathroom gender- binary semaphore
int space;          //total space in bathroom
int space_left;    //space left in bathroom
int curr_gender; //current gender flag- women: 0; men: 1; empty(neutral): 2
int same_wait;   //counter for processes of same gender as bathroom waiting
int opp_wait;      //counter for processes of opposite gender to bathroom waiting
}semaphore_str;
```

The bathroom program does the following tasks:

- Get the command line arguments and set relevant variables
- Initializes the semaphore_str and its variables
- Creates a shared file, writes this structure into it and maps local semaphores to it
- Displays the current usage statistics of the bathroom
- Cleans up the shared file upon receiving an exit (ctrl +c) signal

The enter program- high level overview:

- Get the command line arguments and set relevant variables
- Creates and maps local semaphores to the variables in the shared file created by the bathroom program
- Enter the bathroom: this step might be done in three different ways based on the current status of the bathroom. The process may try to enter while:
    - the bathroom is empty : do a 'try_wait' on the gender semaphore. On success, update the current gender shared variable, acquired the bathroom semaphore using the 'acq_same' function (explained later) and return a '1' to the calling function to let it

know off success. If it fails to acquire the ‘gender’ semaphore, it returns a ‘0’ to the calling function- the calling function has this function try again.
    - the bathroom is occupied by people of the same gender: simply acquire the the bathroom semaphore using the ‘acq_same’ function and return ‘1’ on success, else return ‘0’.
    - the bathroom is occupied by people of the opposite gender: do a ‘try_wait’ on the gender semaphore (the gender semaphore is released by the last person to leave the bathroom). On success, update the current gender shared variable, acquire the bathroom semaphore using the ‘acq_same’ function and return ‘1’ on success. If the ‘try_wait’ failed, the process tries again (mean while, other processes of the same gender have been trying to acquire the gender semaphore).
- Sleep: the process sleeps for the designated interval of time
- Leave the Bathroom: If the process is the last to leave the bathroom, it posts the gender semaphore and sets the current gender to neutral. Finally it posts the bathroom semaphore it holds.

The enter program- finer details:

- Return values: Throughout the program, the following scheme has been used with return values: a ‘1’ indicates success and a ‘0’ indicates a failed attempt and the calling function just lets the process try again at the task, until a ‘1’ is returned. If a ‘-1’ is returned, it indicates the occurrence of something unexpected and the program exits.
- The ‘acq_same’ function: The function takes the gender of the process as an argument and returns success (1) or failure (0) to the calling function. This function takes care of race conditions by using the ‘same_wait’ and ‘opp_wait’ counters (variables shared among the processes) along with the semaphores for gender and bathroom. Every new function increments either of these two counters upon inception (same_wait if the bathroom gender is currently the same as its gender and opp_wait if the bathroom gender is different).
  Supposing a process checks and finds that its gender is the same as the bathroom’s (it believes it can now go ahead and capture the bathroom flag) and gets swapped out of the processor before it actually puts a ‘sem_wait’ on the bathroom. Meanwhile all the other processes of the same gender inside the bathroom may finish sleeping and exit and the gender sign on the bathroom may change. The process to exit the bathroom the last will have no way of knowing that there is one more process of the same gender that has ‘partially entered’ the bathroom. Now if our process wakes, it enters the bathroom without checking, leading to the presence of both sexes in the bathroom simultaneously.
  To handle this possibility, even if a process having the same gender as the bathroom is created at a time when there is room for it in the bathroom, it needs to increment the same_wait counter. Upon successfully entering the bathroom (capturing the bathroom semaphore), this counter is decremented. The last exiting process checks for both the occupancy of the bathroom and the ‘same_wait’ counter before setting the bathroom gender to neutral. Now, if the process under discussion has incremented the same wait counter after checking and finding that the bathroom is of the same gender as itself and goes to sleep before it tries to put a ‘wait’ on the bathroom semaphore, upon waking up, if it finds that the gender has changed, all it needs to do is to decrement the same wait counter and return a ‘0’ to the calling function, so that it can try to gain access to the bathroom again. Hence, the ‘entering the bathroom’ step has become sort of atomic- it can be completely rolled back in case of a failure.

Testing:

The code has been tested for various combinations of invalid inputs (an error message is displayed and the program exits).In order to simulate race conditions, a shell script was used for instantiating various process at precise times.
Also, the code has been checked for large values of bathroom size and number of process trying to use the bathroom.

Evaluation:

What I enjoyed most about this project was the challenge of developing my own protocol for bathroomEnter and bathroomExit. Figuring out all the points in the code where the processes may fail due to other parallely executing (and competing) processes was interesting.
I believe my project satisfies all the requirements of the project as per the specifications. However, the algorithm is not ‘fair’ and ‘starvation free’. These two features could not be included because of the repeating nature of implementation for attempts at semaphores. Given more time, I would have found a way of incorporating these features into the current design.
On the plus side, the program takes into consideration the chances of being swapped out of the processor at any point of execution and hence the critical sections of the code have been made ‘atomic operations’ in a way. I believe that all such parts of the code have been taken into consideration. Also, the code has been sufficiently modularized for ease of readability.